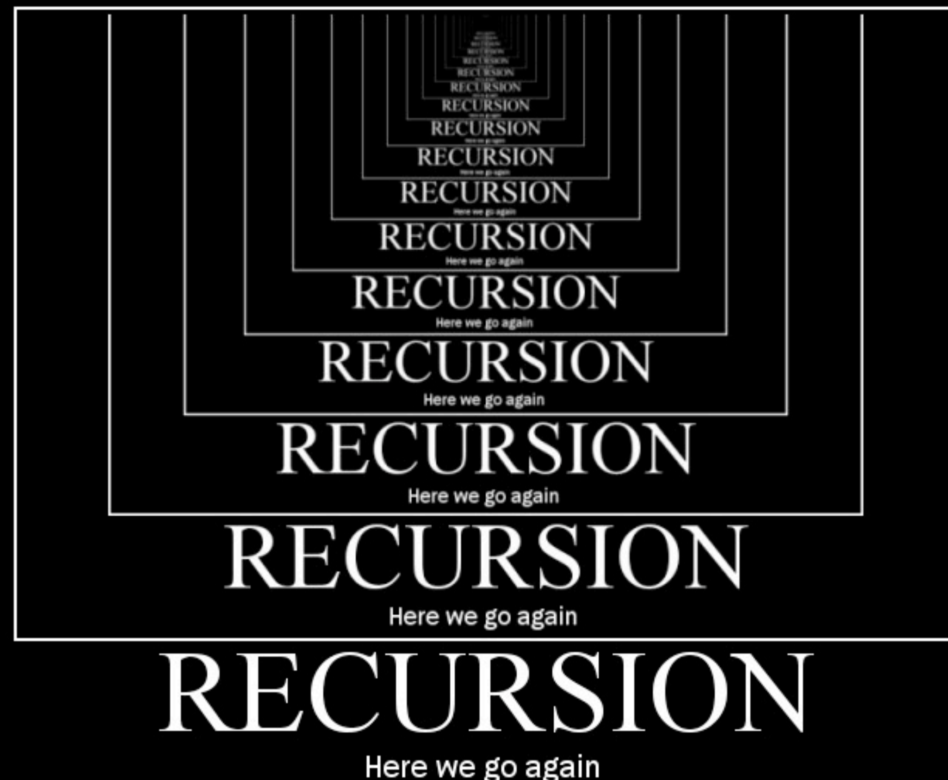
Use what you've got

Recursion

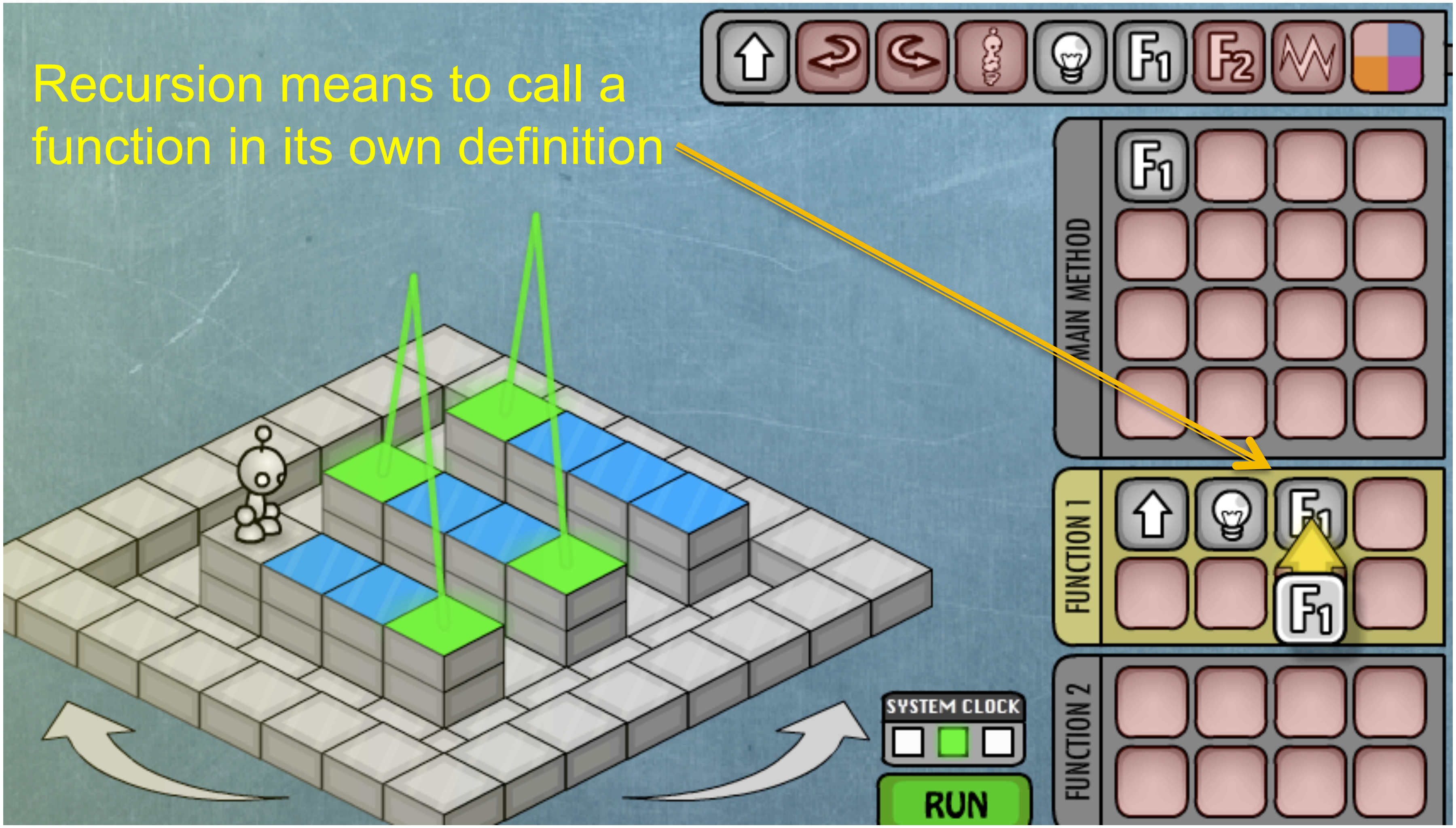


Lawrence Snyder
University of Washington

Announcements

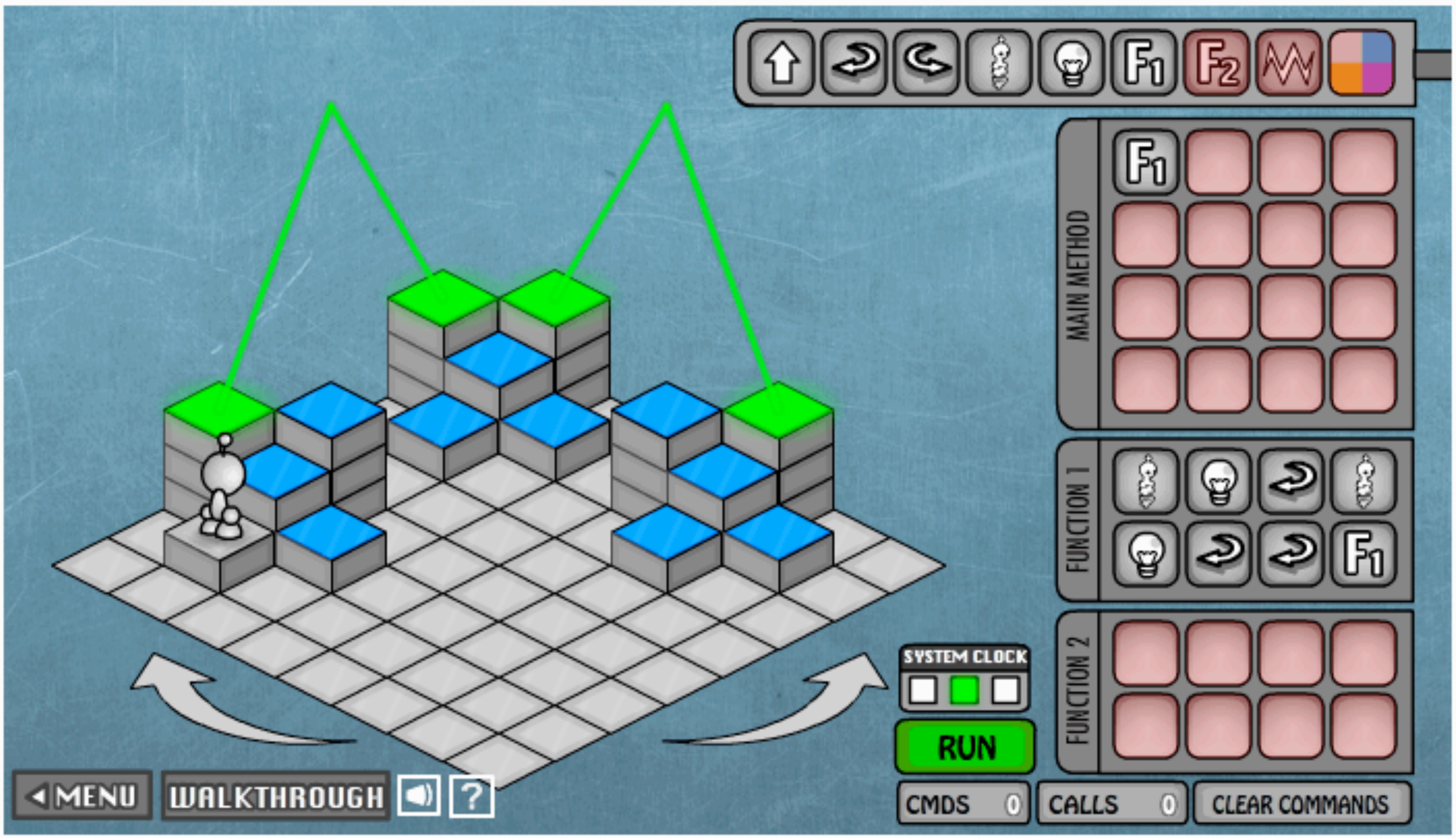
- Upcoming events … it'll be cool
- Availability survey

Recall Recursion In Lightbot 2.0



Recursion

- If the “concept applies,” use it



Recursion

- Recursion means to use a function in its own definition … that is, there is one or more calls to the function in the body

Factorial ($n! = n*(n-1)* \ldots * 1$) is classic example:

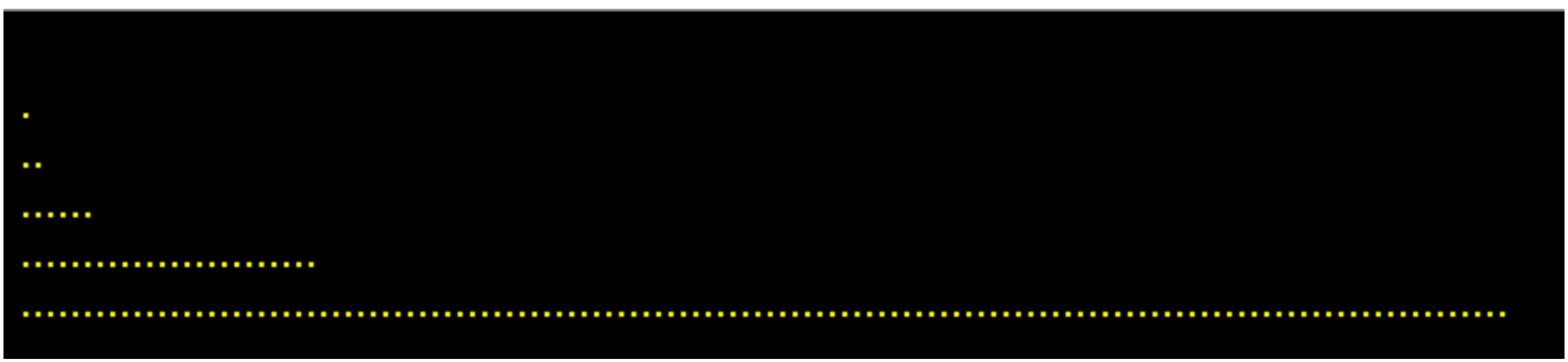
$$fact(n) = \begin{cases} 1 & \text{if } n \text{ is } 0 \text{ or } 1 \\ n*fact(n-1) & \text{otherwise} \end{cases}$$

  - Well – formed recursive functions have two (or more) cases: **basis case**, and a **recursive case**
  - A recursive function must test to separate the "basis case," that is, the non-recursive case, from the normal recursive case

Let's See It In Processing

```
void setup( ) {
  size(700, 200);
  background(0);
  frameRate(4);
  fill(255,255,0);
  for (int i=1; i<6; i++) {
    drawSquare(i);
  }
}

void drawSquare( int n  ) {
  int reps = fact(n);
  for (int i=0; i < reps; i++) {
    rect( 10+i*5, 20+n*20, 3, 3);
  }
}
```

```
int fact (int n) {
  if (n == 1) {
    return 1;
  }
  return n*fact(n-1);
}
```



Recursion is abstraction …

- Recall that when we abstract a rule for a sequence (like drawing 4 squares …

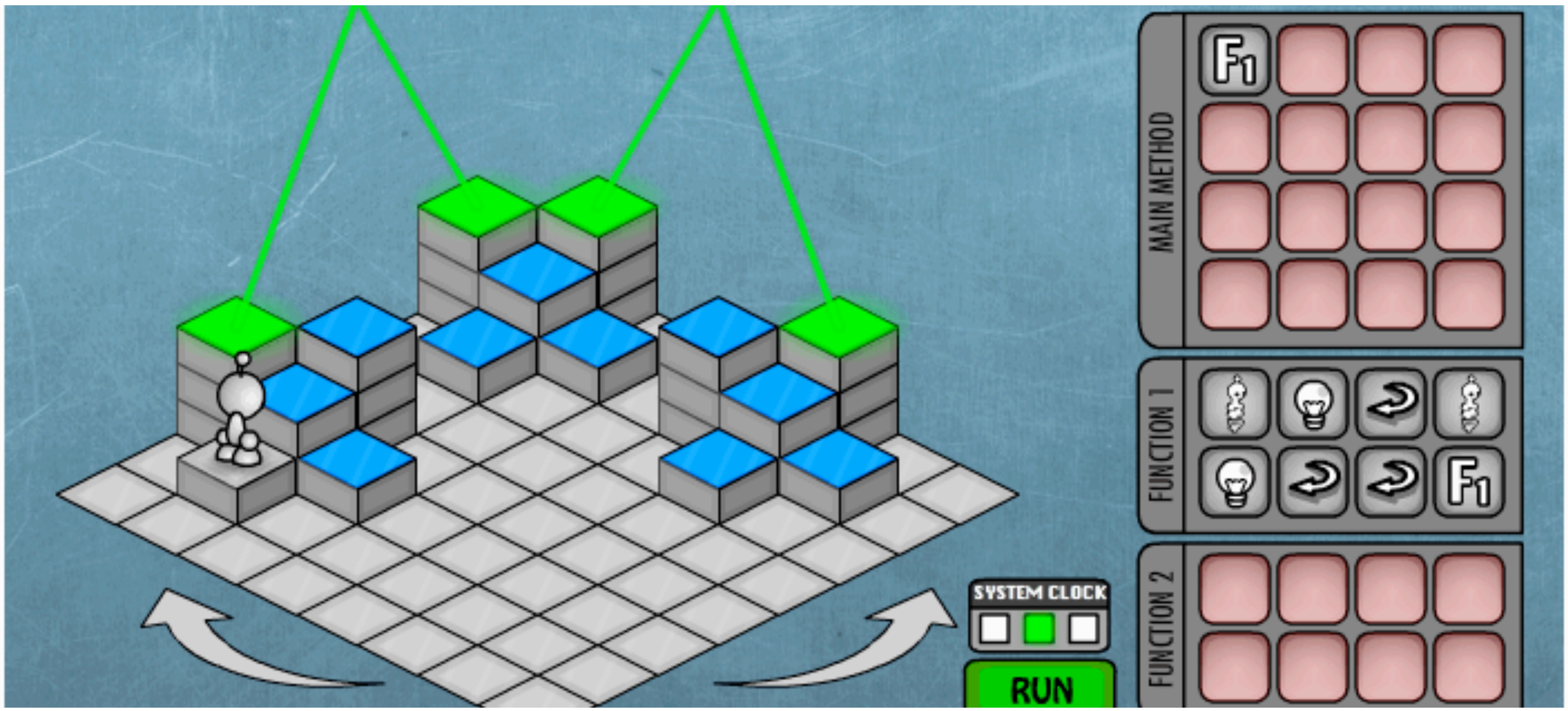
```
for (int i =0; i < 4; i = i+1) {
  rect(100+100*i, 20, 50, 50)
}
```

We try to make each case differ in the same way … allows work to be done automatically)

- Recursion works the same way

Process an "L" corner ...

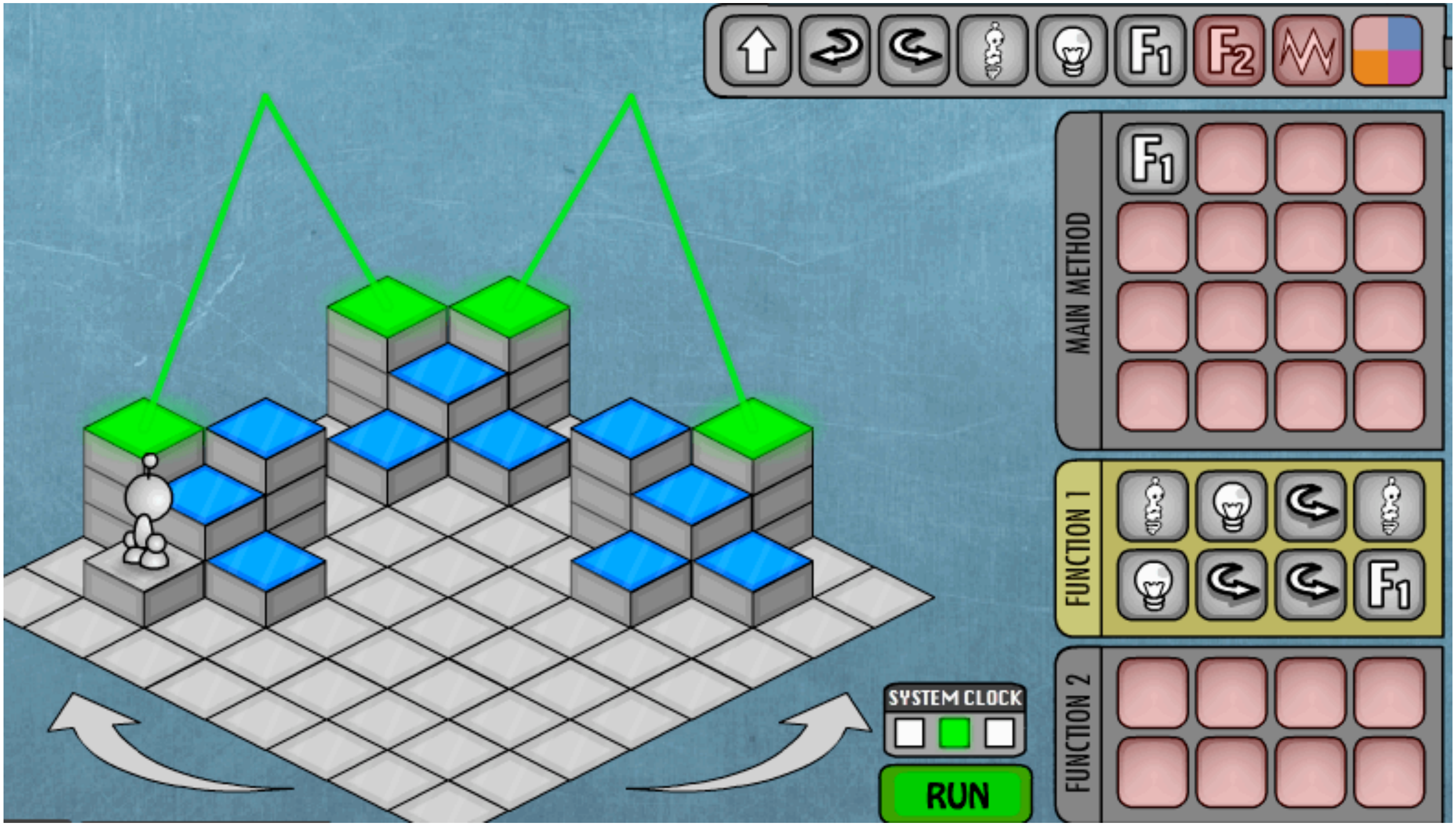
- From its position, Lightbot processes an "L"



- Returns to the same relative position

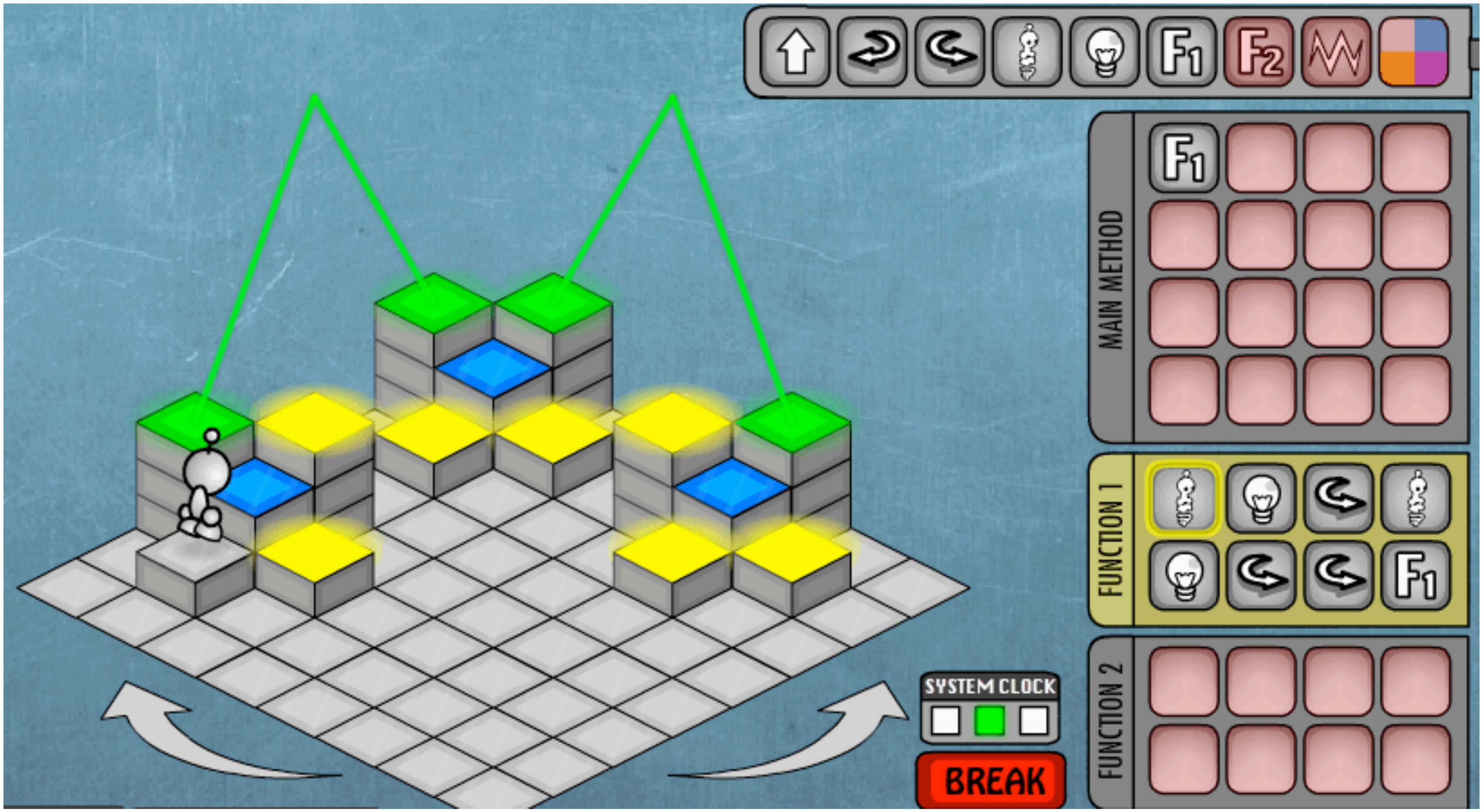
Often Many Approaches Work

- Notice that processing a "7" works, too

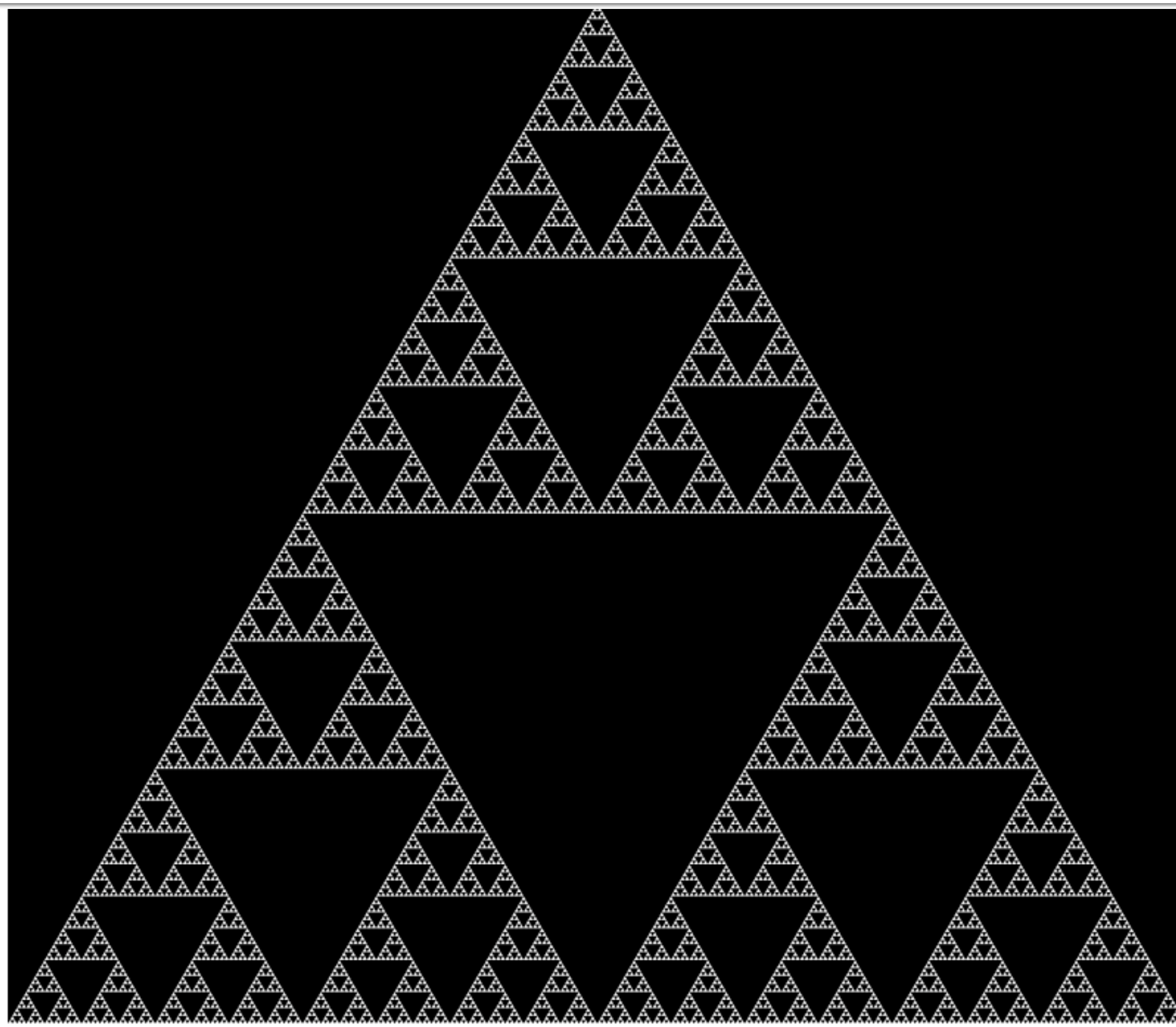


Having Gone Around and Back

- Ready to complete a final circuit



Same Idea: Sierpinski Triangle



Sierpinski Triangle

```
// Sierpinski.pde by Martin Prout
float T_HEIGHT = sqrt(3)/2;
float TOP_Y = 1/sqrt(3);
float BOT_Y = sqrt(3)/6;
float triangleSize = 800;

void setup(){
  size(int(triangleSize),int(T_HEIGHT*triangleSize));
  smooth();
  fill(255);
  background(0);
  noStroke();
  drawSierpinski(width/2, height * (TOP_Y/T_HEIGHT), triangleSize);
}

void drawSierpinski(float cx, float cy, float sz){
  if (sz < 5){  // Limit no of recursions on size
    drawTriangle(cx, cy, sz); // Only draw terminals
    noLoop();
  }
  else{
    float cx0 = cx;
    float cy0 = cy - BOT_Y * sz;
    float cx1 = cx - sz/4;
    float cy1 = cy + (BOT_Y/2) * sz;
    float cx2 = cx + sz/4;
    float cy2 = cy + (BOT_Y/2) * sz;
    drawSierpinski(cx0, cy0, sz/2);
    drawSierpinski(cx1, cy1, sz/2);
    drawSierpinski(cx2, cy2, sz/2);
  }
}

void drawTriangle(float cx, float cy, float sz){
  float cx0 = cx;
  float cy0 = cy - TOP_Y * sz;
  float cx1 = cx - sz/2;
  float cy1 = cy + BOT_Y * sz;
  float cx2 = cx + sz/2;
  float cy2 = cy + BOT_Y * sz;
  triangle(cx0, cy0, cx1, cy1, cx2, cy2);
}
```

Sierpinski Triangle

```
// Sierpinski.pde by Martin Prout
float T_HEIGHT = sqrt(3)/2;
```

```
void setup(){
  size(int(triangleSize),int(T_HEIGHT*triangleSize));
  smooth();
  fill(255);
  background(0);
  noStroke();
  drawSierpinski(width/2, height * (TOP_Y/T_HEIGHT), triangleSize);
}

void drawSierpinski(float cx, float cy, float sz){
  if (sz < 5){  // Limit no of recursions on size
    drawTriangle(cx, cy, sz); // Only draw terminals
    noLoop();
  }
  else{
    float cx0 = cx;
    float cy0 = cy - BOT_Y * sz;
    float cx1 = cx - sz/4;
    float cy1 = cy + (BOT_Y/2) * sz;
    float cx2 = cx + sz/4;
    float cy2 = cy + (BOT_Y/2) * sz;
    drawSierpinski(cx0, cy0, sz/2);
    drawSierpinski(cx1, cy1, sz/2);
    drawSierpinski(cx2, cy2, sz/2);
  }
}
```

```
  triangle(cx0, cy0, cx1, cy1, cx2, cy2);
}
```

Why Recursion Is So Beautiful …

- Often we can solve a problem "top down"
- Finding Fibonacci numbers is classic example –

  1, 1, 2, 3, 5, 8, 13, 21, 34, …

  Each item is the sum of the two before it, except the first two which are both 1
- This definition translates directly:

$$fib(n) = \begin{cases} 1 & \text{if } n<2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- It works like all functions work

Leave The Thinking To The Agent …

$$fib(n) = \begin{cases} 1 & \text{if } n<2 \\ fib(n-1) + fib(n-2) & \text{otherwise} \end{cases}$$

- Compute Fibonacci number 4:
- fib(4) = fib(3) + fib(2)
  - fib(3) = fib(2) + fib(1)
    - fib(2) = fib(1) + fib(0) = 1 + 1 = 2
  - = 2 + 1 = 3
- = 3 + fib(2)
  - fib(2) = fib(1) + fib(0) = 1 + 1 = 2
- = 3 + 2 = 5

Programmers don't need to worry about the details if the definition is right and the termination is right; the computer does the rest

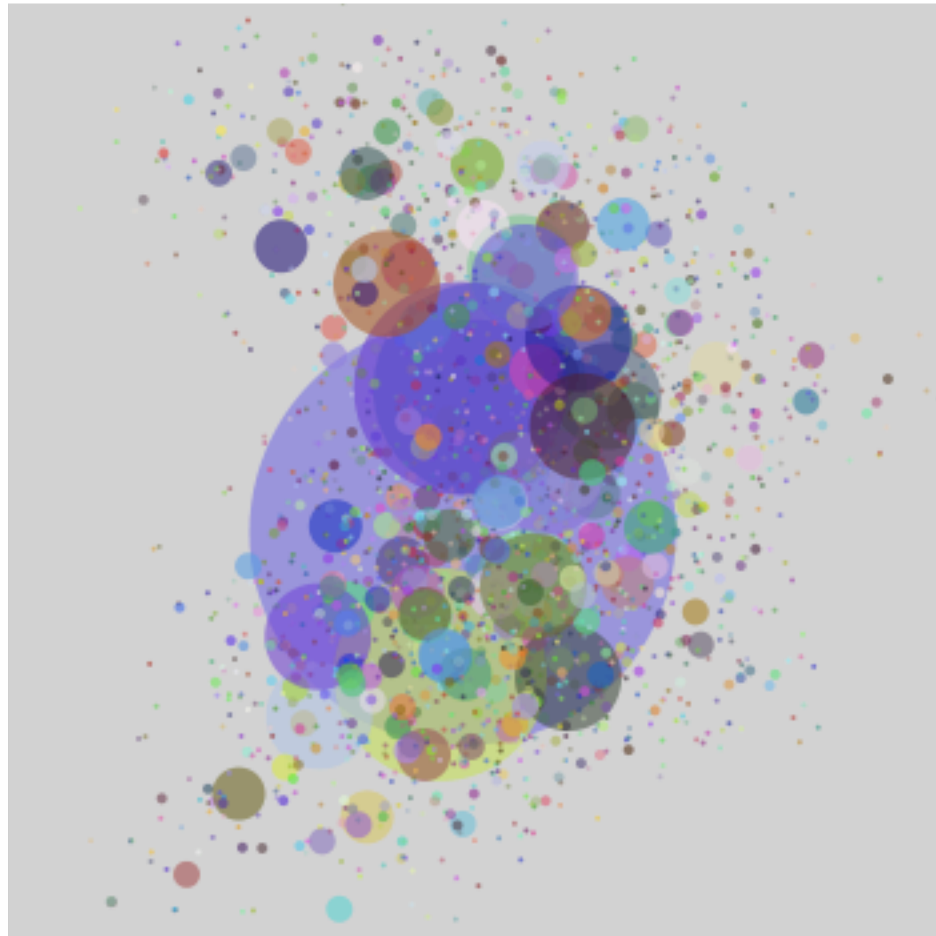
Making It All Work

- Recall that each parameter is created with a function call, and initialized; when the function is over, it is thrown away … which means: "inner" params hide "outer" parameters
- Think about *fact*(4)

```
int fact (int n) {
  if (n == 1) {
    return 1;
  }
  return n*fact(n-1);
}
```

Other Resources … Everywhere

- Recursion is a big deal because it is so elegant; as a result information is everywhere – e.g. see Wikipedia
- See Processing Ref for this cute program



Recursion Is Good For Enumeration

- Give all ways to arrange *k* things in sequences of length *n*
- The key idea is to be orderly about how you do it
- Here we have two kinds of hearts and they are in sequences of 4
  - Notice
    - the top half are red
    - bottom half are white ... recursively

Processing Solution

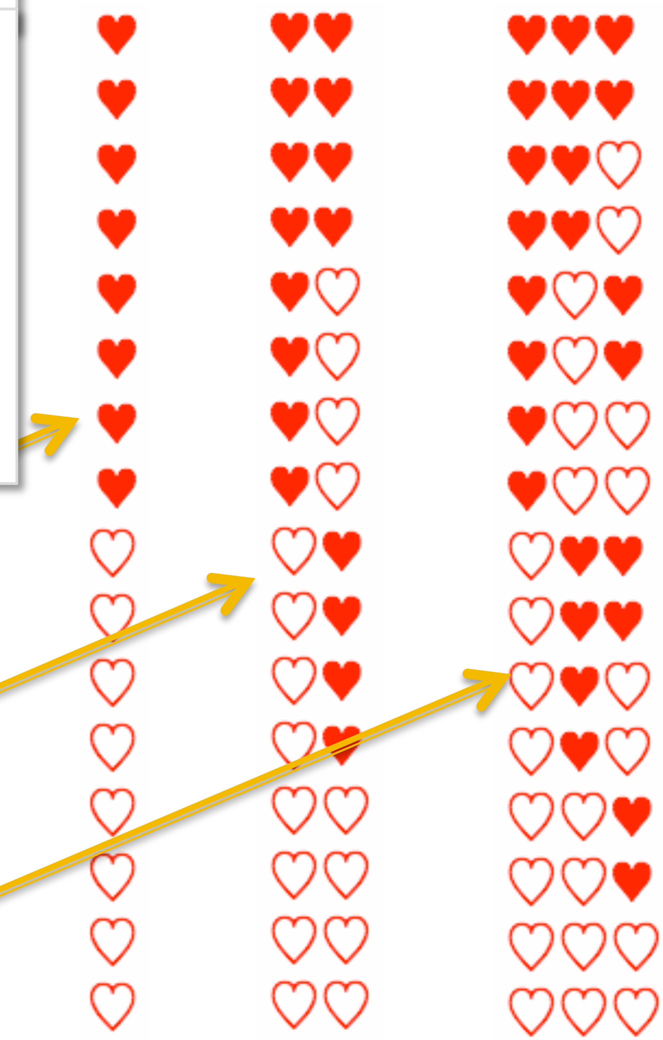
enumerate

```
PFont myFont, mySpecial;
int n = 16;
String[] seq = new String[n];
String[] binDigits = {"♥", "♡"};

void setup( ) {
  size(200, 550);
  myFont = loadFont("EdwardianScriptITC-2
  mySpecial = loadFont("LucidaSans-24.vlw
  smooth();
  fill(255,0,0);
}

void draw( ) {
  background(255);
  smooth();                                  //Prepare for the caption
  textFont(myFont);                          //Caption, line 1
  text("Putting My Heart", 20, 20);          //Caption, line 2
  text("Into Binary", 20, 45);               //Prepare to display UTF-8
  textFont(mySpecial);                       //Initialize with empties
  for (int i=0; i<n; i++) {
    seq[i] = "";
  }
  addon(n, 0, "");                           //Build the String sequences
  for (int j=0; j< n; j++) {                 //And print them
    text(seq[j], 20, 80+j*30);
  }
}
```

```
void addon(int span, int base, String nextdigit) {
  for (int i = 0; i < span; i++) {
    seq[i+base] = seq[i+base] + nextdigit;
  }
  if (span > 1) {
    addon(span/2, base, binDigits[0]);
    addon(span/2, base+span/2, binDigits[1]);
  }
}
```

Watch The Enumeration …

```
void addon(int span, int base, String nextdigit) {
   for (int i = 0; i < span; i++) {
     seq[i+base] = seq[i+base] + nextdigit;
   }
   if (span > 1) {
     addon(span/2, base, binDigits[0]);
     addon(span/2, base+span/2, binDigits[1]);
   }
}
```



A Theoretical Fact

- It is a fact of computer science that loops like our for-loop are unnecessary for programming: All computation can be performed using recursion alone

- It is actually fun and totally cool to program this way!
- In CSP we use recursion when it's convenient

Assignment: Enumerate

- Assignment 14 is to enumerate some strings
- Useful advice –
  - Read the whole assignment before starting
  - Study the solution in these slides, since they are similar